4. Inheritance

OBJEKTIF :

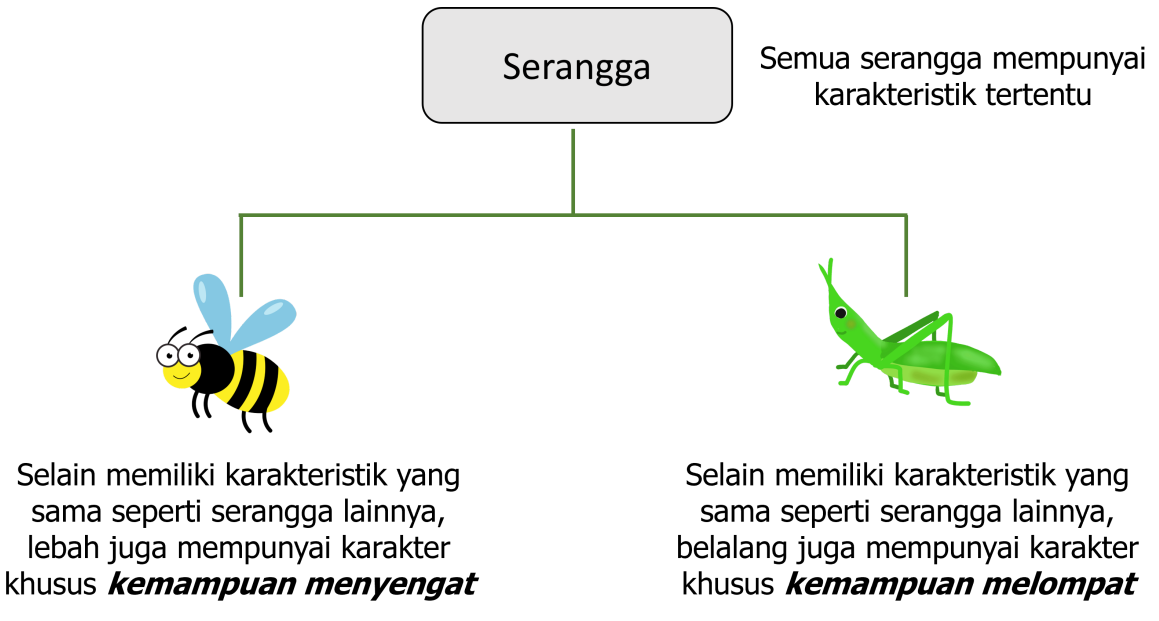
1. Mahasiswa mampu memahami konsep Inheritance.
2. Mahasiswa mampu memahami class hierarchies.
3. Mahasiswa mampu memahami tentang sub classes.
4. Mahasiswa mampu memahami dan melakukan overriding method.
5. Mahasiswa mampu memahami tentang visibility class.

4.1 Class Hierarchies

Salah satu fitur yang paling kuat dalam pemrograman beriorientasi *object* adalah penggunaan kode kembali (*code reuse*). Sekali sebuah *method* dibuat, maka kita bisa menggunakannya berulang kali. Dalam pemrograman beriontasi *object*, kita juga bisa mendesain hubungan antara *class* dengan cara mengelola *class-class* dan faktor kemiripan diantara *class* tersebut. Tujuan utama dari pewarisan atau ***inheritace*** adalah untuk menyediakan fungsionalitas tersebut.

Pada materi sebelumnya, ketika kita membuat beberapa *class*, *class-class* tersebut dapat memiliki beberapa *attribute* dan *method* yang sama. Kita dapat menambahkan *class* baru yang memiliki fitur-fitur yang sama tersebut dan meninggalkan hanya fitur-fitur yang berbeda kepada *class* aslinya. Proses ini disebut sebagai generalisasi dan spesialisasi. Hubungan antara *class* general dan spesialisasinya disebut sebagai ***Inheritance*** atau pewarisan. *Class general* yang sifatnya lebih umum biasanya disebut dengan `superclass` atau *parent class*. Sedangkan *class* spesialisasi yang sifatnya lebih spesifik biasa disebut dengan `subclass` atau *child class*. `superclass` akan mewarisi atau menurunkan segala atribut dan *method* yang ia punya kepada `subclass`. Hal ini akan mengizinkan pembuatan *class* baru yang didasarkan dari pengabstrakan atribut-atribut dan *behaviour* yang sama.

Object yang merupakan versi spesialisasi dari suatu *object* mempunyai sebuah relasi "Is a" (adalah sebuah) dengan *object* yang merupakan versi generalisasinya. Sebagai contoh, perhatikan ilustrasi gambar berikut.

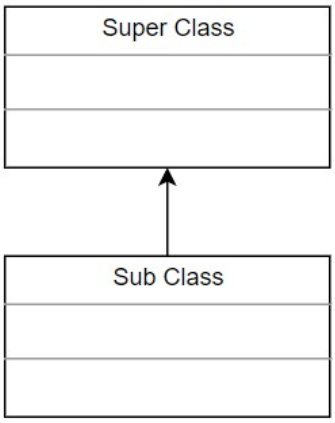


Serangga adalah jenis binatang yang mempunyai ciri-ciri tertentu. Belalang dan lebah adalah binatang-binatang yang termasuk serangga karena mereka mempunyai karakteristik-karakteristik dari serangga. Selain itu, mereka juga mempunyai karakteristik-karakteristik masing-masing. Misalkan, belalang mempunyai kemampuan untuk melompat dan lebah mempunyai kemampuan menyengat. Belalang dan lebah adalah versi spesialisasi dari serangga. Pada ilustrasi gambar di atas, belalang "is a" (adalah seekor) serangga, begitu pula dengan lebah. Contoh-contoh lain relasi "is a":

- Mobil "is a" (adalah sebuah) kendaraan.
- Bunga "is a" (adalah sebuah) tumbuhan.
- Persegi panjang "is a" (adalah sebuah) bentuk.
- Pemain bulu tangkis "is a" (adalah seorang) atlet.

Jika terdapat sebuah relasi "is-a" diantara dua *object*, ini berarti *object* versi spesialisasi mempunyai semua karakteristik dari *object* versi umumnya, dan juga karakteristik tambahan yang membuatnya lebih khusus. Dalam pemrograman berorientasi *object*, *inheritance* (pewarisan) digunakan untuk membuat sebuah relasi "is -a" antara *class-class*. *Inheritance* memungkinkan kita untuk mengekstensi kemampuan dari sebuah *class* dengan membuat *class* lain yang merupakan versi spesialisasi dari *class* tersebut.

Pada diagram UML, diagram hubungan antara *class general* atau `superclass` dengan *class* spesialisasinya atau `subclass` digambarkan dengan anak panah mengarah ke `superclass` seperti berikut.



Sedangkan dalam penerapannya pada kode Java, digunakan kata kunci *extends* untuk membuat suatu `subclass` mewarisi sifat dari `superclass`. Berikut sintaks *class* dengan *inheritance*.

```
[access specifier] class subclass extends superclass
{
    // Class body
}
```

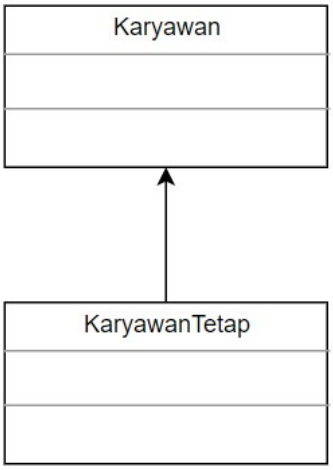


Diagram di atas merupakan contoh pewarisan atau *inheritance*, *class* `Karyawan` menjadi `superclass` yang mewariskan atribut-atributnya serta *method*nya ke *class* `KaryawanTetap` yang mana `KaryawanTetap` adalah `subclass` dari *class* `Karyawan`. Berikut merupakan contoh kode dari diagram diatas.

Definisi Class (Karyawan.java)

```
public class Karyawan
{
    // Class body
}
```

Definisi Class (KaryawanTetap.java)

```
public class KaryawanTetap extends Karyawan
{
    // Class body
}
```

Bisa terlihat bahwa di setiap nama dari `subclass` akan diikuti oleh keyword *extends* dan nama `superclass` yang menandakan bahwa *class* tersebut merupakan `subclass` dari `superclass`.

Saat membuat aplikasi dengan sistem yang sangat besar, kita akan dihadapkan dengan banyaknya *object* yang ada. Serta tiap-tiap *object* tersebut pasti memiliki kesamaan atribut maupun *method* satu sama lain, maka kita bisa mengklasifikasikannya menggunakan ***inheritance***. Dalam *inheritance* terdapat `superclass` dan juga `subclass`, untuk memperjelas perbedaan antara *superclass* dan *subclass* kita harus membuat hirarki dari *class* tersebut.

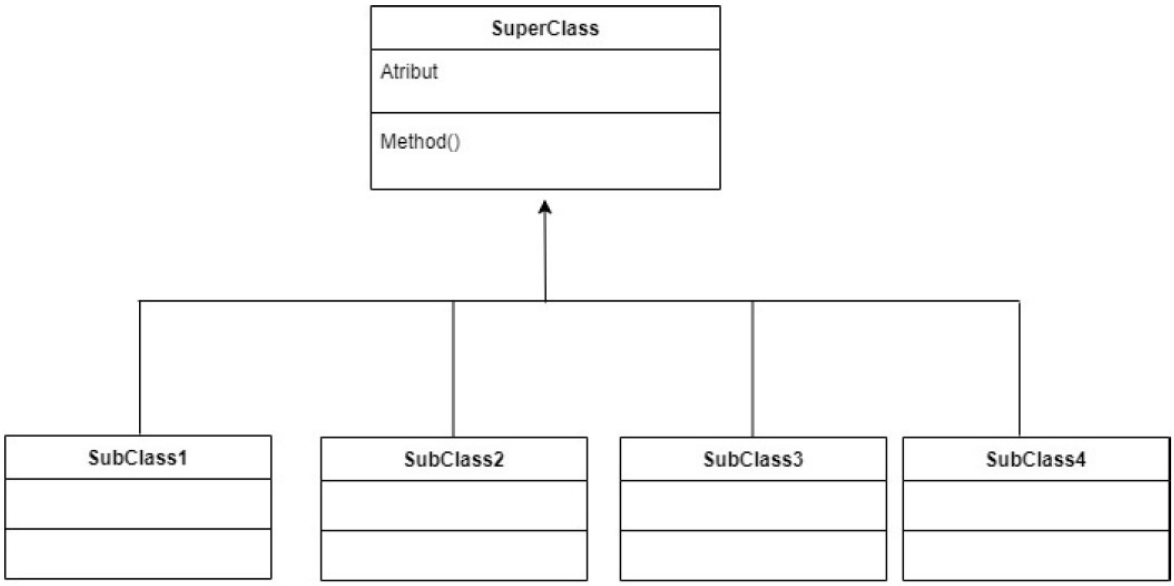


Diagram di atas merupakan gambaran dari *class* hirarki pada *inheritance*, letak `superclass` berada di paling atas sedangkan letak `subclass` berada di bagian bawah `superclass`. Dari hirarki tersebut dapat disimpulkan bahwa yang akan mewariskan segala atribut maupun *method* ialah `superclass` sedangkan yang akan diwariskan adalah `subclass`. Hirarki dari *class* tersebut sama seperti silsilah sebuah keluarga, kita harus mengetahui siapa yang menjadi *parent* (`superclass`) dan siapa yang menjadi *child* (`subclass`).

Dalam *class* hirarki, untuk menunjuk suatu nilai dari sebuah atribut dan memanggil sebuah *method* kita bisa menggunakan *keyword* `this` dan `super`. Masing-masing *keyword* memiliki fungsinya sendiri.

4.1.1 Keyword `this`

Keyword `this` berfungsi sebagai variabel referensi yang dapat digunakan oleh sebuah *object* untuk mereferensikan dirinya sendiri. Pada penerapannya, *keyword* `this` bisa digunakan untuk mereferensikan *field* ataupun untuk memanggil *constructor* yang terdapat dalam *class* yang sama.

Menggunakan `this` untuk Mereferensikan *Field*

Ketika kita menuliskan *method instance* kita harus menggunakan nama variabel parameter yang berbeda dari nama *field*. Sebagai contoh, perhatikan *class* `PersegiPanjang` berikut.

Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    public PersegiPanjang(double pjg, double lbr)
    {
        panjang = pjg;
        lebar = lbr;
    }

    public void setPanjang(double pjg)
    {
        panjang = pjg;
    }
```

```
    public void setLebar(double lbr)
    {
        lebar = lbr;
    }

    public double getPanjang()
    {
        return panjang;
    }

    public double getLebar()
    {
        return lebar;
    }

    public double getLuas()
    {
        return panjang * lebar;
    }
}
```

pada *method* `setPanjang` dari *class* `PersegiPanjang` kita menggunakan nama variabel parameter `pjg` untuk membedakannya dengan field `panjang`:

```
public void setPanjang(double pjg)
{
    panjang = pjg;
}
```

Kita tidak dapat menggunakan nama variabel parameter yang sama dengan nama *field*. Jika kita menggunakan nama variabel parameter yang sama dengan nama *field* kita akan kehilangan akses ke *field* tersebut.

Terkadang sulit dan memerlukan waktu yang tidak sebentar untuk memikirkan nama parameter yang berbeda dengan nama *field*. Untuk menghindari kesulitan ini, kita dapat menggunakan nama parameter yang sama dengan nama *field* yang terkait, dengan menambahkan *keyword* `this` untuk mereferensikan nama *field*. Sebagai contoh, *method* `setPanjang` dapat kita tulis ulang sebagai berikut:

```
public void setPanjang(double panjang)
{
    this.panjang = panjang;
}
```

Menggunakan `this` untuk Memanggil *Constructor* Lain dari Sebuah *Constructor*

Selain untuk mereferensikan *field*, kita dapat juga menggunakan `this` untuk memanggil *constructor* lain dalam sebuah *class*. Pada *class* `PersegiPanjang`, kita mempunyai *constructor* yang menerima dua argumen seperti berikut:

```
public PersegiPanjang(double pjg, double lbr)
{
    panjang = pjg;
    lebar = lbr;
}
```

Misalkan kita ingin menambahkan sebuah *constructor* lain yang menerima semua argumen dan menugaskan nilai argumen tersebut ke *field* `panjang` dan *field* `lebar`. Kita dapat menuliskan *contructor* tersebut seperti berikut:

```
public PersegiPanjang(double sisi)
{
    this(sisi, sisi);
}
```

Constructor di atas menggunakan variabel referensi `this` untuk memanggil *constructor* yang menerima dua argumen. *Constructor* ini memberikan nilai dalam variabel parameter `sisi` sebagai argumen ke parameter `pjg` dan parameter `lbr` dari *constructor* yang menerima dua argumen. Hasil dari pemanggilan *constructor* ini adalah nilai dalam `sisi` ditugaskan ke *field* `panjang` dan *field* `lebar`.

Ketika kita menggunakan `this` untuk memanggil *constructor* lain, kita harus memperhatikan dua aturan berikut:

- `this` hanya dapat digunakan untuk memanggil sebuah *constructor* dari *constructor* lain dalam *class* yang sama.
- *Statement* pemanggilan *constuctor* lain dengan `this` harus dituliskan sebagai *statement* pertama dalam *constructor* yang memanggil. Jika tidak, error kompilasi akan terjadi.

Kode berikut adalah kode class `PersegiPanjang` yang ditulis ulang menggunakan `this` untuk mereferensikan field dan menggunakan `this` untuk memanggil constructor lain:

Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang()
    {
        panjang = 0.0;
        lebar = 0.0;
    }

    public PersegiPanjang(double panjang, double lebar)
    {
        this.panjang = panjang;
        this.lebar = lebar;
    }
```

```
    /*
        Constructor
        @param sisi Panjang dan lebar dari persegi panjang.
    */
    public PersegiPanjang(double sisi)
    {
        this(sisi, sisi);
    }

    public void setPanjang(double panjang)
    {
        this.panjang = panjang;
    }

    public void setLebar(double lebar)
    {
        this.lebar = lebar;
    }

    public double getPanjang()
    {
        return panjang;
    }

    public double getLebar()
    {
        return lebar;
    }

    public double getLuas()
    {
        return panjang * lebar;
    }
}
```

4.1.2 Constructor Superclass

Dalam relasi *inheritance*, *constructor* `superclass` selalu dieksekusi sebelum *constructor* `subclass`. Untuk memperlihatkan *constructor* `superclass` selalu dieksekusi sebelum *constructor* `subclass` dieksekusi saat pembuatan *object* `subclass`, perhatikan dua kode class berikut:

Definisi Class (SuperClass1.java)

```
public class SuperClass1
{
    /*
        Constructor
    */
    public SuperClass1()
    {
        System.out.println("Ini adalah constructor superclass.");
    }
}
```

Definisi Class (SubClass1.java)

```
public class SubClass1 extends SuperClass1
{
    /*
        Constructor
    */
    public SubClass1()
    {
        System.out.println("Ini adalah constructor subclass.");
    }
}
```

Kode pertama adalah kode *class* `SuperClass1` yang mempunyai *constructor* tanpa argumen. *Constructor* ini hanya menampilkan pesan "Ini adalah constructor superclass". Kode kedua adalah kode dari `SubClass1` yang mengekstensi `SuperClass1`. *Class* ini juga mempunyai *constructor* tanpa argumen yang menampilkan pesan "Ini adalah constructor subclass".

Program berikut mendemokan pembuatan object `SubClass1`:

Program (DemoConstructor.java)

```
public class DemoConstructor1
{
    public static void main(String[] args)
    {
        SubClass1 obj = new SubClass1();
    }
}
```

Output Program (DemoConstructor.java)

```
Ini adalah constructor superclass.
Ini adalah constructor subclass.
```

Seperti yang dapat Anda lihat pada *output* program, *constructor* `superclass` dieksekusi terlebih dahulu lalu diikuti oleh *constructor* `subclass`.

Hal yang perlu diingat mengenai *constructor* `superclass` dan *constructor* `subclass` dalam relasi *inheritance* adalah jika sebuah `superclass` memiliki *constructor default* atau *constructor* tanpa argumen yang ditulis dalam *class*, maka *constructor* tersebut akan secara otomatis dipanggil sebelum *constructor* `subclass` dieksekusi. Kita akan melihat situasi-situasi lain yang melibatkan *constructor* `superclass` pada bagian berikutnya.

4.1.3 Keyword `super`

Pada bagian sebelumnya, kita telah melihat contoh-contoh yang menunjukkan bagaimana *constructor default* atau *constructor* tanpa argumen dari `superclass` secara otomatis dipanggil sebelum *constructor* `subclass` dieksekusi. Bagaimana jika `superclass` tidak mempunyai *constructor default* atau *constructor* tanpa argumen? Atau jika `superclass` mempunyai lebih dari satu *constructor* ter-*overloading* dan kita ingin memastikan satu *constructor* tertentu yang dipanggil? Dalam situasi-situasi seperti ini, kita menggunakan *keyword* `super` untuk memanggil *constructor* `superclass` secara eksplisit. *Keyword* `super` merujuk ke `superclass` dari *object* dan dapat digunakan untuk mengakses member-member dari `superclass`.

Untuk memperlihatkan penggunaan *keyword* `super` perhatikan dua kode *class* berikut:

Definisi Class (SuperClass2.java)

```
public class SuperClass2
{
    /*
        Constructor #1
    */
    public SuperClass2()
    {
        System.out.println("Ini adalah constructor tanpa argument superclass");
    }

    /*
        Constructor #2
    */
    public SuperClass2(int arg)
    {
        System.out.println("Argument berikut diberikan " +
                           "ke constructor superclass: " + arg);
    }
}
```

Definisi Class (SubClass2.java)

```
public class SubClass2 extends SuperClass2
{
    /*
        Constructor
    */
    public SubClass2()
    {
        super(10);
        System.out.println("Ini adalah constructor subclass.");
    }
}
```

Perhatikan kode dari `SubClass2`. Pada baris 8 di dalam *constructor* `SubClass2` kita menuliskan *statement* berikut:

```
super(10);
```

Statement ini memanggil *constructor* `superclass` dengan memberikan nilai 10 sebagai argumen. Terdapat tiga ketentuan dalam pemanggilan *constructor* `superclass`:

- *Statement* `super` yang memanggil *constructor* `superclass` hanya dapat dituliskan di dalam *constructor* `subclass`. Kita tidak dapat memanggil *constructor* `superclass` dari *method-method* lainnya.
- *Statement* `super` yang memanggil *constructor* `superclass` haruslah *statement* pertama dalam *constructor* `subclass`. Ini karena *constructor* `superclass` harus dieksekusi sebelum kode dalam *constructor* `subclass` dieksekusi.
- Jika *constructor* `subclass` tidak secara eksplisit memanggil *constructor* `superclass`, Java akan secara otomatis memanggil *constructor default* dari `superclass`, atau memanggil

constructor tanpa argumen dari `superclass` sebelum mengeksekusi kode-kode di dalam *constructor* `subclass`.

Program berikut mendemonstrasikan kedua class di atas:

Program (DemoConstructor2.java)

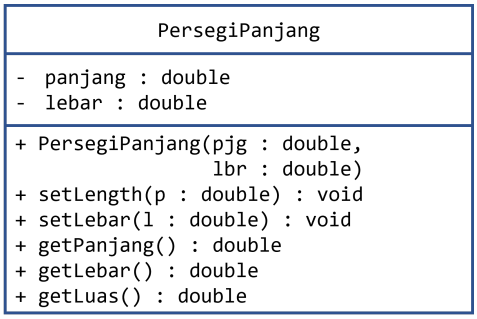
```
/*
    Porgram ini mendemonstrasikan bagaimana constructor
    superclass dipanggil dengan keyword super.
*/
public class DemoConstructor2
{
    public static void main(String[] args)
    {
        SubClass2 obj = new SubClass2();
    }
}
```

Output Program (DemoConstructor2.java)

```
Argument berikut diberikan ke constructor superclass: 10
Ini adalah constructor subclass.
```

Contoh Lain Pemanggilan Constructor Superclass

Kita akan menggunakan *class* `PersegiPanjang` yang kita buat pada topik sebelumnya. Gambar berikut adalah diagram UML dari *class* `PersegiPanjang`:



Berikut adalah potongan dari kode *class* `PersegiPanjang`:

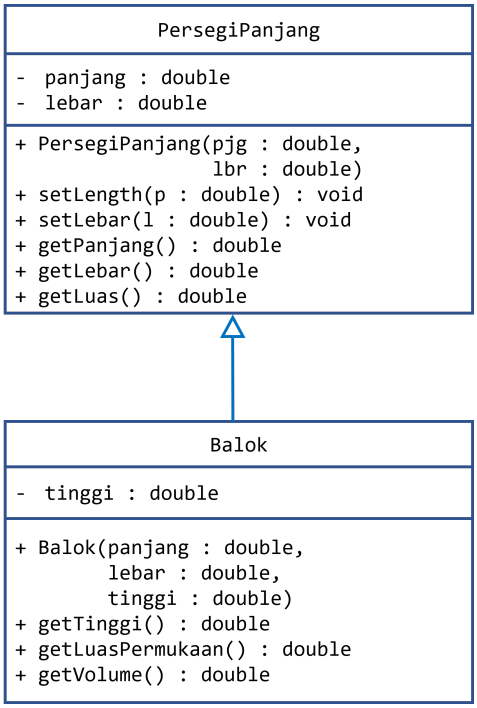
```
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang(double pjg, double lbr)
    {
        panjang = pjg;
```

```
        lebar = lbr;
    }

    ...terdapat kode-kode lain yang tidak ditampilkan disini.
}
```

Selanjutnya kita akan membuat *class* `Balok`, yang mengekstensi *class* `PersegiPanjang`. *Class* `Balok` didesain untuk menyimpan data mengenai balok, yang tidak hanya mempunyai panjang, lebar, dan luas (luas dasar), tapi juga mempunyai tinggi, luas permukaan, dan volume. Gambar berikut adalah diagram UML yang menunjukkan relasi *inheritance* dari *class* `PersegiPanjang` dan *class* `Balok`:



Kode berikut adalah definisi *class* `Balok`:

Definisi Class (Balok.java)

```
public class Balok extends PersegiPanjang
{
    private double tinggi;      // Tinggi balok

    public Balok(double panjang, double lebar, double tinggi)
    {
        // Panggil constructor superclass
        super(panjang, lebar);
        this.tinggi = tinggi;
    }

    public double getTinggi()
    {
        return tinggi;
    }

    /*
        Method getLuasPermukaan menghitung dan
        mengembalikan luas permukaan balok.
        @return Luas permukaan balok.
    */
```

```
    public double getLuasPermukaan()
    {
        return 2 * getLuas() +
               2 * getPanjang() * tinggi +
               2 * getLebar() * tinggi;
    }

    /*
        Method getVolume menghitung dan
        mengembalikan volume balok.
     */
    public double getVolume()
    {
        return getLuas() * tinggi;
    }
}
```

Pada kode di atas, kita mendefinisikan *constructor* `Balok` untuk menerima argumen-argumen untuk parameter-parameter `panjang`, `lebar`, dan `tinggi`. Nilai-nlai yang diberikan ke `panjang` dan `lebar` kita berikan lagi sebagai argumen ke *constructor* `PersegiPanjang` pada baris 15:

```
super(panjang, lebar);
```

Setelah *constructor* `PersegiPanjang` selesai dieksekusi, kode-kode berikutnya dalam *constructor* `Balok` kemudian dieksekusi.

Program berikut mendemonstrasikan *class* `Balok` ini:

Program (DemoBalok.java)

```
import java.util.Scanner;

public class DemoBalok
{
    public static void main(String[] args)
    {
        double panjang;     // Panjang balok
        double lebar;       // Lebar balok
        double tinggi;      // Tinggi balok

        // Buat object Scanner untuk input keyboard
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan dimensi balok.
        System.out.println("Masukkan dimensi dari balok!");
        System.out.print("Panjang: ");
        panjang = keyboard.nextDouble();
        System.out.print("Lebar: ");
        lebar = keyboard.nextDouble();
        System.out.print("Tinggi: ");
        tinggi = keyboard.nextDouble();

        // Buat object Balok dan berikan dimensi ke constructor.
        Balok myBalok = new Balok(panjang, lebar, tinggi);

        // Tampilkan properti-properti balok.
```

```
        System.out.println("Berikut adalah properti-properti dari balok.");
        System.out.println("Panjang = " + myBalok.getPanjang());
        System.out.println("Lebar = " + myBalok.getLebar());
        System.out.println("Tinggi = " + myBalok.getTinggi());
        System.out.println("Luas Permukaan = " + myBalok.getLuasPermukaan());
        System.out.println("Volume = " + myBalok.getVolume());
    }
}
```

Output Program (DemoBalok.java)

```
Masukkan dimensi dari balok!
Panjang: 5
Lebar: 2
Tinggi: 3
Berikut adalah properti-properti dari balok.
Panjang = 5.0
Lebar = 2.0
Tinggi = 3.0
Luas Permukaan = 62.0
Volume = 30.0
```

Hal yang perlu dicatat pada contoh di atas adalah class `PersegiPanjang` mempunyai satu *constructor* yang menerima dua argumen. Karena terdapat *constructor* pada *class* `PersegiPanjang` maka Java tidak menyediakan secara otomatis *constructor default*. Sehingga, pada *class* `PersegiPanjang` tidak terdapat *constructor* tanpa argumen ataupun *constructor default*. Jika `superclass` tidak mempunyai *constructor default* atau *constructor* tanpa argumen, maka *class* yang mengekstensinya harus memanggil salah satu *constructor* yang dipunyai oleh `superclass`. Jika tidak, error kompilasi akan didapatkan.

Ringkasan Hal-Hal Penting Mengenai Constructor pada Inheritance

Berikut adalah ringkasan hal-hal penting yang perlu kita ketahui mengenai *constructor* pada *inheritance*:

- *Constructor* `superclass` selalu dieksekusi sebelum *constructor* `subclass`.
- Kita dapat menuliskan *statement* `super` yang memanggil *constructor* `superclass`, tetapi hanya pada *constructor* `subclass`. Kita tidak dapat memanggil *constructor* `superclass` pada *method-method* lain.
- Jika *statement* `super` memanggil *constructor* `superclass` terdapat di dalam *constructor* `subclass`, *statement* tersebut harus sebagai *statement* pertama.
- Jika *constructor* `subclass` tidak secara eksplisit memanggil *constructor* `superclass`, Java akan secara otomatis memanggil `super()` sebelum mengeksekusi kode-kode di dalam *constructor* `subclass`.
- Jika `superclass` tidak mempunyai *constructor default* maupun *constructor* tanpa argumen, maka *class* yang mengekstensinya harus memanggil salah satu *constructor* yang dimiliki oleh `superclass`.

Hal penting dalam *inheritance* ini adalah untuk sebisa mungkin menggunakan *method* yang sudah dideklarasikan didalam `superclass`, karena tujuan kita melakukan *inheritance* adalah agar kita tidak perlu menulis semua atribut maupun *method* yang sama berulang kali. Lalu dalam *class hierarchy*, kita bisa menggunakan *keyword* `this` untuk mereferensikan *field* ataupun untuk memanggil *constructor* yang terdapat dalam *class* yang sama. Sedangkan untuk memanggil

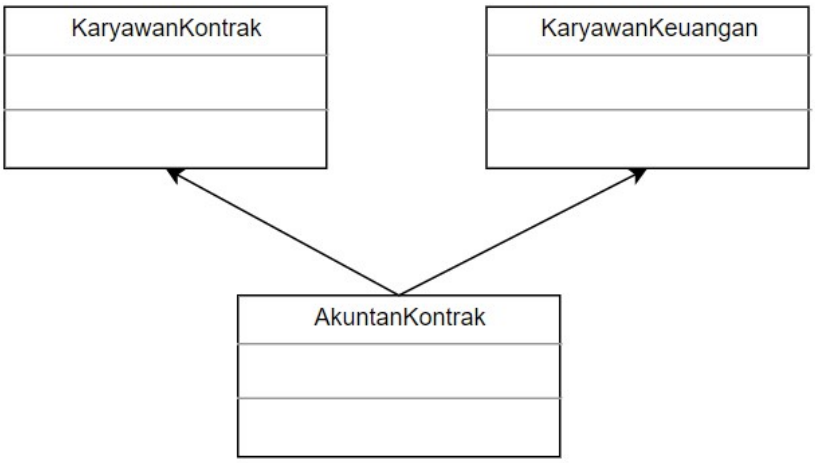
constructor `superclass` secara eksplisit dan dapat digunakan untuk mengakses member-member dari `superclass`, kita bisa menggunakan *keyword* `super`.

4.2 Sub Classes

Saat kita mengumpulkan *class* kedalam *class hierarcy*, kita bisa membagi kode-kode yang umum pada setiap *class*. Dalam Java, kita membuat `subclass` dengan memberi atribut atau *method* khusus yang membedakan antara `subclass` dengan `superclass`. Object `subclass` otomatis akan memiliki segala atribut-atribut dan *method-method* yang telah didefinisikan didalam `superclass`. Dalam `subclass` kita hanya perlu mendeklarasikan atribut atau method khusus yang tidak ada pada `superclass`. Terdapat beberapa karakteristik dari `subclass`, yaitu :

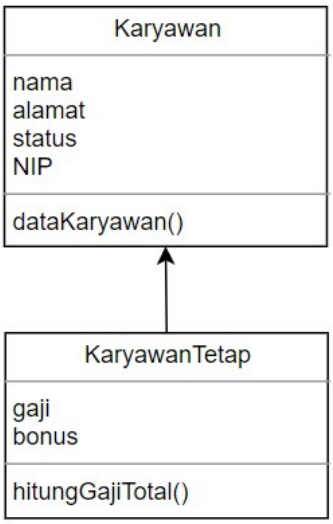
- `Subclass` merupakan bentuk khusus dari sebuah `superclass`.
- Sebuah `subclass` memiliki atribut-atribut dan *method-method* yang diturunkan dari sebuah `superclass`. Atribut-atribut dan *method-method* yang diturunkan dari sebuah `superclass` dapat langsung digunakan di dalam `subclass`.
- `Subclass` bisa memiliki atribut-atribut dan *methods* tambahan yang belum ada di `superclass`nya.
- Terminologi *super* dan *sub* berasal dari teori himpunan. Himpunan objek dari `subclass` adalah *subset* dari semua himpunan objek `superclass`. Dan semua himpunan objek dari `superclass` merupakan *superset* dari himpunan `subclass`.

Pemrograman menggunakan Java hanya memperbolehkan *single inheritance* saja, yaitu setiap satu *class* hanya dapat mempunyai sebuah `superclass` dan dapat memiliki banyak `subclass` dan dapat terjadi secara bertingkat. Java tidak mengizinkan adanya pewarisan sifat lebih dari satu *class* (memiliki banyak `superclass`) atau yang biasa disebut dengan *multiple inheritance* dengan alasan dapat membingungkan.



Gambar di atas merupakan contoh *Multiple Inheritance* yang tidak diizinkan oleh Java. Berdasarkan diagram, dapat dilihat bahwa *class* `AkuntanKontrak` memiliki lebih dari satu `superclass` atau lebih dari satu pewarisan sifat. Hal tersebutlah yang tidak diperbolehkan dalam Java karena akan menyebabkan kebingungan ketika nantinya program akan dijalankan.

4.2.1 *Single Inheritance* dengan Satu `subclass`



Gambar di atas merupakan contoh *single inheritance* dengan satu `subclass` yang mana *superclass*nya adalah `Karyawan` dan *subclass*nya adalah `KaryawanTetap`. Atribut dan *method* pada *class* `Karyawan` akan diwariskan ke *class* `KaryawanTetap`. Berikut merupakan contoh penerapannya.

Definisi Class (Karyawan.java)

```
public class Karyawan
{
    // Attributes
    private String nama;
    private String alamat;
    private String status;
    private int NIP;

    // Constructor
    public Karyawan(String nama, String alamat, String status, int NIP)
    {
        this.nama = nama;
        this.alamat = alamat;
        this.status = status;
        this.NIP = NIP;
    }

    public String getNama()
    {
        return this.nama;
    }

    public void setNama(String nama)
    {
        this.nama = nama;
    }

    public String getAlamat()
    {
        return this.alamat;
    }
```

```
    public void setAlamat(String alamat)
    {
        this.alamat = alamat;
    }

    public String getStatus()
    {
        return this.status;
    }

    public void setStatus(String status)
    {
        this.status = status;
    }

    public int getNIP()
    {
        return this.NIP;
    }

    public void setNIP(int NIP)
    {
        this.NIP = NIP;
    }

    // Method dataKaryawan() untuk menampilkan nilai atribut dari class karyawan
    public void dataKaryawan()
    {
        System.out.println("=========Data karyawan=========\n"+
        "Nama : "+nama+"\nAlamat : "+alamat+"\nNIP :"+NIP+
        "\nStatus : "+status);
    }
}
```

Definisi Class (KaryawanTetap.java)

```
public class KaryawanTetap extends Karyawan
{
    // Attributes
    private int gaji;
    private int bonus;

    // Constructor
    public KaryawanTetap(String nama, String alamat, String status,
                         int NIP, int gaji, int bonus)
    {
        super(nama, alamat, status, NIP);
        this.gaji = gaji;
        this.bonus = bonus;
    }

    public int getGaji()
    {
        return this.gaji;
    }

    public void setGaji(int gaji)
```

```
    {
        this.gaji = gaji;
    }

    public int getBonus()
    {
        return this.bonus;
    }

    public void setBonus(int bonus)
    {
        this.bonus = bonus;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal()
    {
        System.out.println("Total Gaji : "+(gaji + bonus));
    }

}
```

Program (Main.java)

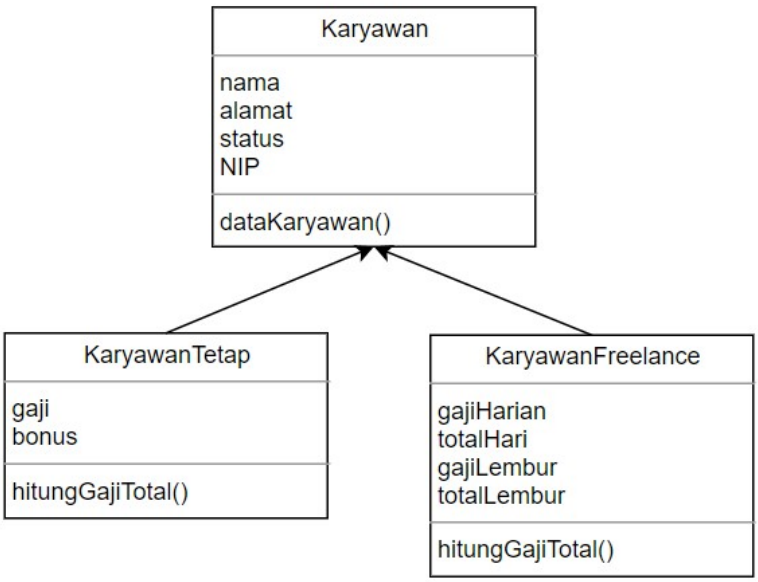
```
public class Main
{
    public static void main(String[] args)
    {
        KaryawanTetap karyawan = new KaryawanTetap("Andini Sari",
                                 "Jl. Margonda Raya", "Karyawan Tetap",
                                 987654321, 4000000, 1500000);
        karyawan.dataKaryawan();
        karyawan.hitungGajiTotal();
    }
}
```

Output Program (Main.java)

```
=========Data karyawan=========
Nama : Andini Sari
Alamat : Jl. Margonda Raya
NIP :987654321
Status : Karyawan Tetap
Total Gaji : 5500000
```

Pada contoh tersebut dapat kita ketahui bahwa atribut dan *method* dari *superclass* yaitu `Karyawan` diwariskan ke *subclass* yaitu `KaryawanTetap`. Dapat dilihat pada *class* `Main`, *object* `karyawan` yang merupakan *object* dari *class* `KaryawanTetap` dapat memanggil *method* `dataKaryawan()` yang mana *method* tersebut berada pada *class* `Karyawan`, bukan `KaryawanTetap`. Hal tersebut dapat dilakukan karena *class* `Karyawan` telah mewariskan *method* dan atributnya ke *class* `KaryawanTetap`.

4.2.2 *Single Inheritance* dengan Banyak `subclass`



Gambar di atas merupakan contoh *single inheritance* dengan banyak `subclass` yang mana *superclass*nya adalah `Karyawan` dan *subclass*nya adalah `KaryawanTetap` dan `KaryawanFreelance`. Atribut dan *method* pada *class* `Karyawan` akan diwariskan ke *class* `KaryawanTetap` dan `KaryawanFreelance`. Sehingga *class* `KaryawanTetap` dan *class* `KaryawanFreelance` memiliki kemiripan sifat tetapi kedua *class* tersebut tidak memiliki hubungan satu sama lain. Berikut merupakan contoh penerapannya.

Definisi Class (Karyawan.java)

```
public class Karyawan
{
    // Attributes
    private String nama;
    private String alamat;
    private String status;
    private int NIP;

    // Constructor
    public Karyawan(String nama, String alamat, String status, int NIP)
    {
        this.nama = nama;
        this.alamat = alamat;
        this.status = status;
        this.NIP = NIP;
    }

    public String getNama()
    {
        return this.nama;
    }

    public void setNama(String nama)
    {
        this.nama = nama;
    }

    public String getAlamat()
```

```
    {
        return this.alamat;
    }

    public void setAlamat(String alamat)
    {
        this.alamat = alamat;
    }

    public String getStatus()
    {
        return this.status;
    }

    public void setStatus(String status)
    {
        this.status = status;
    }

    public int getNIP()
    {
        return this.NIP;
    }

    public void setNIP(int NIP)
    {
        this.NIP = NIP;
    }

    // Method dataKaryawan() untuk menampilkan nilai atribut dari class karyawan
    public void dataKaryawan()
    {
        System.out.println("=========Data karyawan=========\n"+
        "Nama : "+nama+"\nAlamat : "+alamat+"\nNIP :"+NIP+
        "\nStatus : "+status);
    }
}
```

Definisi Class (KaryawanTetap.java)

```
public class KaryawanTetap extends Karyawan
{
    // Attributes
    private int gaji;
    private int bonus;

    // Constructor
    public KaryawanTetap(String nama, String alamat, String status,
                         int NIP, int gaji, int bonus)
    {
        super(nama, alamat, status, NIP);
        this.gaji = gaji;
        this.bonus = bonus;
    }

    public int getGaji()
    {
```

```
        return this.gaji;
    }

    public void setGaji(int gaji)
    {
        this.gaji = gaji;
    }

    public int getBonus()
    {
        return this.bonus;
    }

    public void setBonus(int bonus)
    {
        this.bonus = bonus;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal()
    {
        System.out.println("Total Gaji : "+(gaji + bonus));
    }
}
```

Definisi Class (KaryawanFreelance.java)

```
public class KaryawanFreelance extends Karyawan
{
    // Attributes
    private int gajiHarian;
    private int totalHari;
    private int gajiLembur;
    private int totalLembur;

    // Constructor
    public KaryawanFreelance(String nama, String alamat, String status, int NIP,
                             int gajiHarian, int totalHari, int gajiLembur,
                             int totalLembur)
    {
        super(nama, alamat, status, NIP);
        this.gajiHarian = gajiHarian;
        this.totalHari = totalHari;
        this.gajiLembur = gajiLembur;
        this.totalLembur = totalLembur;
    }

    public int getGajiHarian()
    {
        return this.gajiHarian;
    }

    public void setGajiHarian(int gajiHarian)
    {
        this.gajiHarian = gajiHarian;
    }
```

```
    public int getTotalHari()
    {
        return this.totalHari;
    }

    public void setTotalHari(int totalHari)
    {
        this.totalHari = totalHari;
    }

    public int getGajiLembur()
    {
        return this.gajiLembur;
    }

    public void setGajiLembur(int gajiLembur)
    {
        this.gajiLembur = gajiLembur;
    }

    public int getTotalLembur()
    {
        return this.totalLembur;
    }

    public void setTotalLembur(int totalLembur)
    {
        this.totalLembur = totalLembur;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal(){
        System.out.println("Total Gaji : "+
                           ((gajiHarian * totalHari) +
                            (gajiLembur * totalLembur)));
    }
}
```

Program (Main.java)

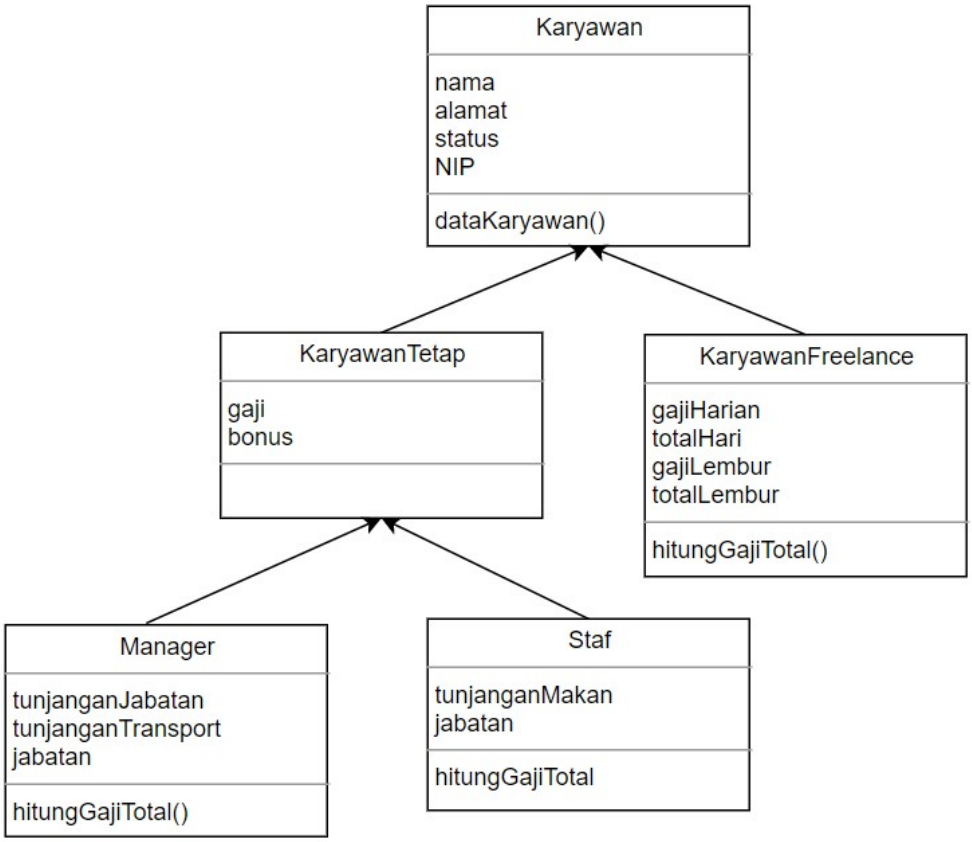
```
public class Main
{
    public static void main(String[] args)
    {
        KaryawanTetap karyawan1 = new KaryawanTetap("Andini Sari",
                                  "Jl. Margonda Raya", "Karyawan Tetap",
                                  987654321, 4000000, 1500000);
        karyawan1.dataKaryawan();
        karyawan1.hitungGajiTotal();

        KaryawanFreelance karyawan2 = new KaryawanFreelance("Rino Angkasa",
                                      "Jl. Salemba Raya","Karyawan Freelance",
                                      123456789, 200000, 15, 50000, 10);
        karyawan2.dataKaryawan();
        karyawan2.hitungGajiTotal();
    }
}
```

Output Program (Main.java)

```
=========Data karyawan=========
Nama : Andini Sari
Alamat : Jl. Margonda Raya
NIP :987654321
Status : Karyawan Tetap
Total Gaji : 5500000
=========Data karyawan=========
Nama : Rino Angkasa
Alamat : Jl. Salemba Raya
NIP :123456789
Status : Karyawan Freelance
Total Gaji : 3500000
```

4.2.3 *Single Inheritance* Bertingkat



Gambar di atas merupakan contoh *single inheritance* bertingkat. Perhatikan pada *class* `KaryawanTetap`, *class* tersebut menjadi *subclass* dari *superclass* `Karyawan` tetapi juga menjadi *superclass* yang mewarisi sifat ke *subclass* `Manager` dan `Staf`. Hal inilah yang menyebabkan dinamakannya dengan *single inheritance* bertingkat. Atribut dan *method* pada *class* `Karyawan` akan diwariskan ke *class* `KaryawanTetap` dan `KaryawanFreelance` dan juga Atribut dan *method* pada *class* `KaryawanTetap` akan diwariskan ke *class* `Manager` dan `Staf` . Sehingga *class* `KaryawanTetap` dan *class* `KaryawanFreelance` memiliki kemiripan sifat tetapi kedua *class* tersebut tidak memiliki hubungan satu sama lain, begitu pula berlaku untuk *class* `Manager` dan *class* `Staf`. Berikut merupakan contoh penerapannya.

Definisi Class (Karyawan.java)

```
public class Karyawan
{
```

```
    // Attributes
    private String nama;
    private String alamat;
    private String status;
    private int NIP;

    // Constructor
    public Karyawan(String nama, String alamat, String status, int NIP)
    {
        this.nama = nama;
        this.alamat = alamat;
        this.status = status;
        this.NIP = NIP;
    }

    public String getNama()
    {
        return this.nama;
    }

    public void setNama(String nama)
    {
        this.nama = nama;
    }

    public String getAlamat()
    {
        return this.alamat;
    }

    public void setAlamat(String alamat)
    {
        this.alamat = alamat;
    }

    public String getStatus()
    {
        return this.status;
    }

    public void setStatus(String status)
    {
        this.status = status;
    }

    public int getNIP()
    {
        return this.NIP;
    }

    public void setNIP(int NIP)
    {
        this.NIP = NIP;
    }

    // Method dataKaryawan() untuk menampilkan nilai atribut dari class karyawan
    public void dataKaryawan()
    {
```

```
        System.out.println("=========Data karyawan=========\n"+
        "Nama : "+nama+"\nAlamat : "+alamat+"\nNIP :"+NIP+
        "\nStatus : "+status);
    }
}
```

Definisi Class (KaryawanFreelance.java)

```
public class KaryawanFreelance extends Karyawan
{
    // Attributes
    private int gajiHarian;
    private int totalHari;
    private int gajiLembur;
    private int totalLembur;

    // Constructor
    public KaryawanFreelance(String nama, String alamat, String status, int NIP,
                             int gajiHarian, int totalHari, int gajiLembur,
                             int totalLembur)
    {
        // mengambil nilai superclassnya yaitu karyawan
        super(nama, alamat, status, NIP);
        this.gajiHarian = gajiHarian;
        this.totalHari = totalHari;
        this.gajiLembur = gajiLembur;
        this.totalLembur = totalLembur;
    }

    public int getGajiHarian()
    {
        return this.gajiHarian;
    }

    public void setGajiHarian(int gajiHarian)
    {
        this.gajiHarian = gajiHarian;
    }

    public int getTotalHari()
    {
        return this.totalHari;
    }

    public void setTotalHari(int totalHari)
    {
        this.totalHari = totalHari;
    }

    public int getGajiLembur()
    {
        return this.gajiLembur;
    }

    public void setGajiLembur(int gajiLembur)
    {
        this.gajiLembur = gajiLembur;
```

```
    }

    public int getTotalLembur()
    {
        return this.totalLembur;
    }

    public void setTotalLembur(int totalLembur)
    {
        this.totalLembur = totalLembur;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal(){
        System.out.println("Total Gaji : "+
                           ((gajiHarian * totalHari) +
                            (gajiLembur * totalLembur)));
    }
}
```

Definisi Class (KaryawanTetap.java)

```
public class KaryawanTetap extends Karyawan
{
    // Attributes
    private int gaji;
    private int bonus;
    // Constructor
    public KaryawanTetap(String nama, String alamat, String status,
                         int NIP, int gaji, int bonus)
    {
        // mengambil nilai superclass nya yaitu karyawan
        super(nama, alamat, status, NIP);
        this.gaji = gaji;
        this.bonus = bonus;
    }
    public int getGaji()
    {
        return this.gaji;
    }

    public void setGaji(int gaji)
    {
        this.gaji = gaji;
    }

    public int getBonus()
    {
        return this.bonus;
    }

    public void setBonus(int bonus)
    {
        this.bonus = bonus;
    }
}
```

Definisi Class (Manager.java)

```
public class Manager extends KaryawanTetap
{
    // Attributes
    private int tunjanganJabatan;
    private int tunjanganTransport;
    private String jabatan;
    // Constructor
    public Manager(String nama, String alamat, String status, int NIP,
                   int gaji, int bonus, int tunjanganJabatan,
                   int tunjanganTransport, String jabatan)
    {
        // Mengambil nilai superclass nya yaitu KaryawanTetap
        super(nama, alamat, status, NIP, gaji, bonus);
        this.tunjanganJabatan = tunjanganJabatan;
        this.tunjanganTransport = tunjanganTransport;
        this.jabatan = jabatan;
    }

    public int getTunjanganJabatan()
    {
        return this.tunjanganJabatan;
    }
    public void setTunjanganJabatan(int tunjanganJabatan)
    {
        this.tunjanganJabatan = tunjanganJabatan;
    }

    public int getTunjanganTransport()
    {
        return this.tunjanganTransport;
    }

    public void setTunjanganTransport(int tunjanganTransport)
    {
        this.tunjanganTransport = tunjanganTransport;
    }

    public String getJabatan()
    {
        return this.jabatan;
    }

    public void setJabatan(String jabatan)
    {
        this.jabatan = jabatan;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal()
    {
        System.out.println("Jabatan : "+jabatan+"\nTotal Gaji : "+
                           (super.getGaji() + super.getBonus() +
                           tunjanganJabatan + tunjanganTransport));
    }
}
```

Definisi Class (Staf.java)

```
public class Staf extends KaryawanTetap
{
    // Attributes
    private int tunjanganMakan;
    private String jabatan;

    // Constructor
    public Staf(String nama, String alamat, String status, int NIP,
                int gajj, int bonus, int tunjanganMakan, String jabatan)
    {
        // Mengambil nilai superclassnya yaitu KaryawanTetap
        super(nama, alamat, status, NIP, gajj, bonus);
        this.tunjanganMakan = tunjanganMakan;
        this.jabatan = jabatan;
    }

    public int getTunjanganMakan()
    {
        return this.tunjanganMakan;
    }
    public void setTunjanganMakan(int tunjanganMakan)
    {
        this.tunjanganMakan = tunjanganMakan;
    }

    public String getJabatan()
    {
        return this.jabatan;
    }
    public void setJabatan(String jabatan)
    {
        this.jabatan = jabatan;
    }

    // Method untuk menghitung total gaji
    public void hitungGajiTotal()
    {
        System.out.println("Jabatan : "+jabatan+"\nTotal Gaji : "+
                           (super.getGaji() + super.getBonus()
                           + tunjanganMakan));
    }
}
```

Program (Main.java)

```
public class Main
{
    public static void main(String[] args)
    {
        KaryawanFreelance karyawan1 = new KaryawanFreelance("Rino Angkasa",
                                      "Jl. Salemba Raya","Karyawan Freelance",
                                      123456789, 200000, 15, 50000, 10);
        karyawan1.dataKaryawan();
        karyawan1.hitungGajiTotal();
```

```
        Manager manager = new Manager("Andini Sari", "Jl. Margonda Raya",
                          "Karyawan Tetap", 987654321, 5000000, 1500000,
                          3000000, 1000000, "Manager");
        manager.dataKaryawan();
        manager.hitungGajiTotal();

        Staf staf = new Staf("Randy Perkasa", "Jl. Cimahi", "Karyawan Tetap",
                    654321789, 4000000, 1000000, 900000, "Staf");
        staf.dataKaryawan();
        staf.hitungGajiTotal();
    }
}
```

Output Program (Main.java)

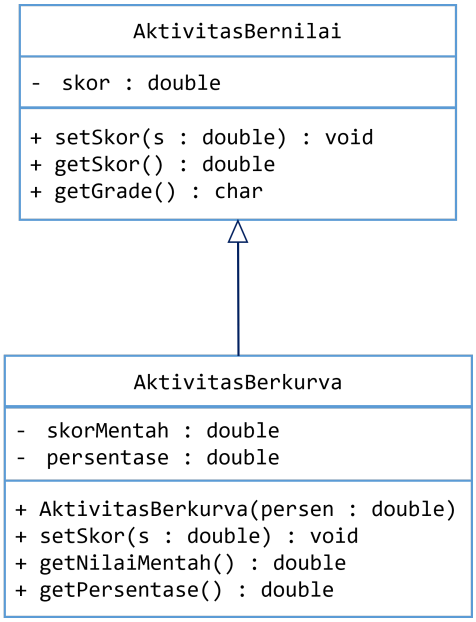
```
=========Data karyawan=========
Nama : Rino Angkasa
Alamat : Jl. Salemba Raya
NIP :123456789
Status : Karyawan Freelance
Total Gaji : 3500000
=========Data karyawan=========
Nama : Andini Sari
Alamat : Jl. Margonda Raya
NIP :987654321
Status : Karyawan Tetap
Jabatan : Manager
Total Gaji : 10500000
=========Data karyawan=========
Nama : Randy Perkasa
Alamat : Jl. Cimahi
NIP :654321789
Status : Karyawan Tetap
Jabatan : Staf
Total Gaji : 5900000
```

4.3 Overriding Method

`subclass` mewarisi *method-method* dari `superclass`. Jika dari *method-method* yang diwarisi terdapat beberapa *method-method* yang tidak cukup atau tidak cocok untuk tujuan dari `subclass`, kita dapat menggantikan *method-method* yang tidak cocok ini dengan meng-*overriding* (menimpa) *method-method* tersebut. Untuk meng-*overriding* suatu method `superclass`, kita menuliskan *method* dengan daftar parameter yang sama dengan *method* `superclass` yang ingin di-*overriding*.

Kita bisa mereplikasi atau membuat *method* di `subclass` dengan nama yang sama seperti yang ada di `superclass` dengan catatan atribut maupun *method* yang ada di `superclass` tidak harus bersifat private. *Class* `superclass` akan mewariskan semua *method public*nya. Kita bisa mendeklarasikan *method* yang baru pada `subclass` lalu mengganti implementasi dari *method* yang diwariskan jika implementasi dari *method* yang diwariskan tidak sesuai atau tidak cukup untuk tujuan dari `subclass` yang kita buat.

Sebagai contoh, *class* `AktivitasBernilai` mempunyai *method* `setSkor` yang menetapkan skor numerik dan *method* `getGrade` yang mengembalikan *grade* huruf berdasarkan skor numerik tersebut. Misalkan, seorang guru ingin memberikan nilai dengan kurva untuk suatu ujian sebelum nilai huruf ditentukan. Nilai dengan kurva ini dihitung dengan mengalikan setiap skor ujian siswa dengan suatu persentase. Lalu nilai yang telah dikalikan dengan suatu persentase ini digunakan untuk menentukan *grade* huruf. Untuk mengakomodasi ini, kita mendesain sebuah *class* baru, `AktivitasBerkurva` yang mengekstensi *class* `AktivitasBernilai` dan mempunyai versi spesialisasi sendiri dari *method* `setSkor`. *Method* `setSkor` dalam `subclass` meng-*overriding* *method* `setSkor` dalam `superclass`. Gambar berikut adalah diagram UML yang menunjukkan relasi antara *class* `AktivitasBernilai` dan *class* `AktivitasBerkurva`:



Definisi Class (AktivitasBernilai.java)

```
public class AktivitasBernilai
{
    private double skor;    // Skor numerik

    public void setSkor(double s)
    {
        skor = s;
    }

    public double getSkor()
    {
        return skor;
    }

    public char getGrade()
    {
        char gradeHuruf;

        if (skor >= 90)
        {
            gradeHuruf = 'A';
        }
        else if (skor >= 80)
        {
            gradeHuruf = 'B';
        }
```

```
        else if (skor >= 70)
        {
            gradeHuruf = 'C';
        }
        else if (skor >= 60)
        {
            gradeHuruf = 'D';
        }
        else
        {
            gradeHuruf = 'E';
        }

        return gradeHuruf;
    }
}
```

Definisi Class (AktivitasBerkurva.java)

```
public class AktivitasBerkurva extends AktivitasBernilai
{
    private double nilaiMentah;     // Skor sebenarnya
    private double persentase;      // Persentase kurva

    /*
        Constructor menetapkan persentase kurva pada field persentase
        dan menetapkan nilai 0.0 ke field nilaiMentah.
        @param persen Persentase kurva.
    */
    public AktivitasBerkurva(double persen)
    {
        persentase = persen;
        nilaiMentah = 0.0;
    }

    /*
        Method setSkor meng-overriding method setSkor superclass.
        Versi ini menerima skor mentah sebagai argument. Skor tersebut
        lalu dikalikan dengan persentase kurva dan hasilnya diberikan sebagai
        argument ke method setSkor dari superclass.
        @param s Skor mentah.
    */
    @Override
    public void setSkor(double s)
    {
        nilaiMentah = s;
        super.setSkor(nilaiMentah * persentase);
    }

    public double getNilaiMentah()
    {
        return nilaiMentah;
    }

    public double getPersentase()
    {
        return persentase;
```

```
    }
}
```

Perhatikan pada baris 28 sampai dengan 33, kita mempunyai kode seperti berikut:

```
@Override
public void setSkor(double s)
{
    nilaiMentah = s;
    super.setSkor(nilaiMentah * persentase);
}
```

Sebelum kita membahas mengenai `@Override` pada baris 28, kita akan melihat definisi *method* `setSkor` pada baris 29 sampai dengan 33 terlebih dahulu. Pada baris 29, header dari *method* `setSkor` kita tuliskan seperti berikut:

```
public void setSkor(double s)
```

Header method `setSkor` dalam *class* `AktivitasBerkurva` mempunyai daftar parameter:

```
setSkor(double)
```

Daftar parameter *method* ini sama dengan daftar parameter *method* `setSkor` dalam *class* `AkvitasBernilai`. Sehingga, *method* `setSkor` dalam *class* `AktivitasBerkurva` ini meng-*overriding method* `setSkor` dalam *class* `AktivitasBernilai`. *Method* ini menerima sebuah argumen bertipe `double` yang merupakan skor mentah. Kemudian, pada baris 31, *statement*:

```
nilaiMentah = s;
```

menugaskan nilai yang diterima oleh parameter `s` ke *field* `nilaiMentah`. Lalu, pada baris 32, terdapat *statement* berikut:

```
super.setSkor(nilaiMentah * persentase);
```

Keyword `super` pada *statement* ini merujuk ke *object* `superclass`. Sehingga, *statement* ini memanggil *method* `setSkor` versi `superclass` dengan memberikan argumen berupa hasil dari ekspresi `nilaiMentah * persentase`. Kita memanggil *method* `setSkor` dari `superclass` karena kita ingin menyimpan skor ini pada *field* `skor` dari `superclass`. Dan karena *field* `skor` adalah *private* maka `subclass` tidak bisa mengakses langsung *field* ini, sehingga kita harus menggunakan *method* `setSkor` dari `superclass`.

Sebelum definisi *method* `setSkor` yang meng-*overriding method* `setSkor` *superclass*nya, pada baris 29, kita menuliskan `@Override`. `@Override` adalah anotasi override. Anotasi ini memberitahukan *compiler* Java bahwa *method* yang dituliskan setelahnya yaitu *method* `setSkor`, dimaksudkan untuk meng-*overriding* sebuah *method* dalam `superclass`.

Anotasi `@Override` tidak harus dituliskan, tetapi dianjurkan untuk ditulis. Dengan menuliskan anotasi ini, jika *method* yang ditulis setelahnya gagal meng-*overriding method* dalam `superclass`, *compiler* akan menampilkan error. Sebagai contoh, misalkan kita salah menuliskan *header* dari *method* pada baris 29 seperti berikut:

```
public void setskor(double s)
```

Jika kita perhatikan seksama nama dari *method* pada *header* di atas, kita akan melihat bahwa nama *method* ini dituliskan dalam huruf kecil semua. Nama ini tidak cocok dengan nama *method* dalam `superclass` yang ingin kita *overriding*, yaitu `setSkor`. Tanpa menuliskan anotasi `@Override`, kode *class* dengan *header method* seperti di atas akan berhasil dikompilasi dan dieksekusi, tetapi kita tidak akan mendapatkan hasil yang sesuai dengan yang kita inginkan karena *method* dalam `subclass` tersebut tidak meng-*overriding method* dalam `superclass`. Namun, dengan menuliskan anotasi `@Override`, *compiler* akan memberikan pesan error yang memberitahukan kita bahwa *method* `subclass` tidak meng-*overriding method* apapun dalam *method* `superclass`.

Program (DemoAktivitasBerkurva.java)

```
import java.util.Scanner;

public class DemoAktivitasBerkurva
{
    public static void main(String[] args)
    {
        double skor;            // Skor mentah
        double persenKurva;     // Persentase Kurva

        // Buat object Scanner untuk membaca input keyboard
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan skor mentah ujian.
        System.out.print("Masukkan skor mentah siswa: ");
        skor = keyboard.nextDouble();

        // Dapatkan persentase kurva
        System.out.print("Masukkan persentase kurva: ");
        persenKurva = keyboard.nextDouble();

        // Buat object AktivitasBerkurva
        AktivitasBerkurva ujianNilaiKurva = new AktivitasBerkurva(persenKurva);

        // Tetapkan skor ujian
        ujianNilaiKurva.setSkor(skor);

        // Tampilkan nilai mentah
        System.out.println("Nilai mentah = " +
                           ujianNilaiKurva.getNilaiMentah() +
                           " poin.");

        // Tampilkan nilai kurva
        System.out.println("Nilai kurva = " +
                           ujianNilaiKurva.getSkor());

        // Tampilkan grade huruf ujian
        System.out.println("Grade ujian = " +
                           ujianNilaiKurva.getGrade());
    }
}
```

Output Program (DemoAktivitasBerkurva.java)

```
Masukkan skor mentah siswa: 87
Masukkan persentase kurva: 1.06
Nilai mentah = 87.0 poin.
Nilai kurva = 92.22
Grade ujian = A
```

Program di atas menggunakan variabel `ujianNilaiKurva` untuk mereferensikan sebuah *object* `AktivitasBerkurva`. Pada baris 29, *statement* berikut digunakan untuk memanggil *method* `setSkor`:

```
ujianNilaiKurva.setSkor(skor);
```

Karena `ujianNilaiKurva` mereferensikan sebuah *object* `AktivitasBerkurva`, *statement* ini memanggil *method* `setSkor` dari *class* `AktivitasBerkurva`, bukan *method* versi *superclass*nya.

Perlu diingat bahwa semua atribut dan *method* yang bersifat *private* pada `superclass` tidak akan bisa diturunkan ke `subclass`, karena segala hal yang bersifat *private* tidak bisa diakses oleh *class* lain terkecuali *class* itu sendiri. Yang berarti bahwa hanya `superclass` yang mampu mengakses atribut dan *method private*nya, sedangkan `subclass` tidak akan bisa mengaksesnya.

Method *Override* memperluas fungsionalitas dari *method* yang sudah diwariskan oleh `superclass`, yang berarti `subclass` yang melakukan *method Override* bisa menghilangkan, merubah maupun menambahkan implementasi pada *method* yang sedang di *Override*. Pada kasus lain, proses *Override* yang terjadi pada `subclass` dapat merubah total fungsionalitas dari *method* yang diwariskan oleh `superclass`.

4.3.1 Perbedaan *Overloading* dan *Overriding*

Overloading dan *overriding* merupakan dua hal yang berbeda. *Overloading* merupakan keadaan dimana ketika lebih dari satu *method* mempunyai nama yang sama tetapi memiliki daftar parameter yang berbeda. Sedangkan *overriding* merupakan keadaan dimana method `subclass` memiliki daftar parameter yang sama dengan *method* `superclass`.

Sebelumnya juga kita telah mengetahui bahwa *method* yang ter-*overloading* dapat berada di dalam *class* yang sama, method *overloading* pada `subclass` juga dapat meng-*overload method* yang berada di `superclass`. Sedangkan *method overriding* tidak bisa meng-*override method* di *class* yang sama. Berikut merupakan ringkasan perbedaan dari *overloading* dan *overriding*.

- Jika dua *method* memiliki nama sama namun daftar parameter yang berbeda, mereka disebut ter-*overloading*. *Method overloading* dapat dilakukan dalam *class* yang sama atau untuk *method* yang berada di `superclass` dan *method* lain yang berada di dalam `subclass`.
- Jika sebuah *method* dalam `subclass` mempunyai daftar parameter yang sama seperti sebuah *method* dalam `superclass`, *method* `subclass` ini disebut meng-*overriding method* `superclass`.

Perbedaan antara *overloading* dan *overriding* ini penting untuk diketahui. Ketika sebuah *method* dalam `subclass` meng-*overloading method* dalam `superclass`, kedua *method* tersebut dapat dipanggil dengan *object* `subclass`. Namun, ketika *method* dalam `subclass` meng-*overriding method* dalam `superclass`, hanya *method* versi `subclass` yang dapat dipanggil dengan *object* `subclass`. Sebagai contoh, perhatikan kode class `SuperClass3` berikut:

Definisi Class (SuperClass3.java)

```
public class SuperClass3
{
    /*
        Method ini menampilkan sebuah int.
        @param arg Sebuah int.
    */
    public void tampilkanNilai(int arg)
    {
        System.out.println("SUPERCLASS: " +
                           "Argument int adalah " + arg);
    }

    /*
        Method ini menampilkan sebuah String.
        @param arg Sebuah String.
    */
    public void tampilkanNilai(String arg)
    {
        System.out.println("SUPERCLASS: " +
                           "Argument String adalah " + arg);
    }
}
```

Class `SuperClass3` di atas mempunyai *method* ter-*overloading* bernama `tampilkanNilai`. Salah satu *method* `tampilkanNilai` menerima sebuah argumen `int` dan *method* `tampilkanNilai` lainnya menerima sebuah argumen `String`. Sekarang perhatikan kode *class* `SubClass3` berikut yang mengekstensi *class* `SuperClass3`:

Definisi Class (SubClass3.java)

```
public class SubClass3 extends SuperClass3
{
    /*
        Method ini meng-overriding sebuah method dalam superclass.
        @param arg Sebuah int.
    */
    @Override
    public void tampilkanNilai(int arg)
    {
        System.out.println("SUBCLASS: " +
                           "Argument int adalah " + arg);
    }

    /*
        Method ini meng-overloading method superclass.
        @param arg Sebuah doble.
    */
    public void tampilkanNilai(double arg)
    {
        System.out.println("SUBCLASS: " +
                           "Argument double adalah " + arg);
    }
}
```

Perhatikan bahwa `SubClass3` juga mempunyai dua *method* bernama `tampilkanNilai`. *Method* yang pertama, pada baris 8 sampai dengan 12, menerima sebuah argumen `int`. *Method* ini meng-*overriding* salah satu *method* dalam `superclass` karena keduanya mempunyai daftar parameter yang sama. *Method* yang kedua, pada baris 18 sampai dengan 22, menerima sebuah argumen `double`. *Method* ini meng-*overloading method* `tampilkanNilai` karena mempunyai daftar parameter yang berbeda dengan *method* `tampilkanNilai` lainnya. Meskipun sekarang terdapat total empat *method* `tampilkanNilai` dalam *subclass* `SubClass3` dan *superclass* `SuperClass33`, hanya tiga *method* yang dapat dipanggil dari object `SubClass3`. Program berikut mendemonstrasikan ini:

Program (DemoTampilkanNilai.java)

```
/*
    Program ini mendemonstrasikan method-method dalam
    class SuperClass3 dan SubClass3.
*/
public class DemoTampilkanNilai
{
    public static void main(String[] args)
    {
        // Buat object SubClass3
        SubClass3 myObject = new SubClass3();

        myObject.tampilkanNilai(10);        // Berikan sebuah int.
        myObject.tampilkanNilai(1.2);       // Berikan sebuah double.
        myObject.tampilkanNilai("Halo");    // Berikan sebuah String.
    }
}
```

Output Program (DemoTampilkanNilai.java)

```
SUBCLASS: Argument int adalah 10
SUBCLASS: Argument double adalah 1.2
SUPERCLASS: Argument String adalah Halo
```

Ketika argumen `int` diberikan ke pemanggilan *method* `tampilkanNilai`, *method* yang berada dalam `subclass` yang dipanggil karena *method* ini meng-*overriding method* dalam `superclass`. Untuk memanggil *method* `superclass` yang di-*overriding*, kita harus menggunakan *keyword* `super` pada *method* di `subclass`. Berikut adalah contohnya:

```
public void tampilkanNilai(int arg)
{
    super.tampilkanNilai(arg);    // Memanggil method superclass.
    System.out.println("SUBCLASS: Argument int adalah " + arg);
}
```

4.3.2 Mencegah Method di-Overriding

Kita dapat mencegah *method* yang berada di `superclass` kita di-*overriding* oleh `subclass` dengan menambahakan keyword `final` pada *header method* tersebut. Berikut merupakan contoh menambahkan `final` pada *header method*.

```
public final void tampilkanNilai(int arg)
```

Jika `subclass` mencoba untuk meng-*override method* tersebut, maka *compiler* akan menampilkan pesan *error*.

4.4 Visibility

Dalam program Java, tiap *class* bisa saling berhubungan satu dengan yang lainnya dengan cara memberi akses terhadap atribut dan *method* mereka. *Inheritance* merupakan salah satu cara menghubungkan *class*. Semua hal yang ada di dalam *class* seperti atribut dan *method* disebut sebagai member. Umumnya pendefinisian *class* ada tingkatan akses yang disebut *visibility* atau *access modifier* atau bisa juga disebut sebagai *access specifier*.

Pada hubungan *class* menggunakan *inheritance*, semua member yang ada dalam `superclass` dapat diakses oleh `subclass`, kecuali member tersebut diberikan *visibility* `private`. *Visibility* tidak hanya bisa diberikan kepada member sebuah *class*, tetapi dapat diberikan kepada *class* itu sendiri. Berikut merupakan contoh dari *visibility*.

```
public class Sepeda {
    private String modelSepeda;

    protected void menjalankanSepeda(){
        System.out.println("Sepeda digowes");
        System.out.println("Sepeda pun berjalan");
    }

    public void detilSepeda(){
        System.out.printf("Model sepeda adalah : %s", modelSepeda);
    }

    void infoSepeda(){
        System.out.println("Ini merupakan class Sepeda");
    }
}
```

Tanpa Modifier

Secara umum ada 3 *visibility* dalam Java, yaitu: `public`, `protected`, dan `private`. Apabila kita tidak menggunakan salah satu dari tiga *keyword* tersebut, maka member atau *class* itu tidak menggunakan *visibility* atau bisa disebut ***no-modifier***. *No-modifier* merupakan *default visibility* karena sebenarnya terdapat *modifier*, namun tidak tertulis. Istilah *default visibility* lebih umum digunakan dibanding *no-modifier*.

Masing - masing *visibility* akan menentukan di mana saja member bisa diakses. Berikut merupakan tabel jangkauan untuk masing - masing *visibility*.

Modifier	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
`default`	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Keterangan:

World artinya seluruh *package* dalam aplikasi

Pada tabel di atas bisa kita ketahui bahwa untuk memberikan akses kepada setiap member atau *class* yang mana member atau *class* tersebut dapat diakses dimana saja, kita bisa menggunakan *visibility* `public`. Sedangkan jika kita tidak memberi modifier maka member dan *class* kita hanya bisa diakses di *class* dan *package* yang sama.

REFERENSI:

[1] Horstmann, Cay S. 2012. *Big Java: Late Objects, 1st Edition*. United States of America: John Wiley & Sons, Inc.

[2] Gaddis, Tony. 2016. *Starting Out with Java: From Control Structures through Objects (6th Edition)*. Boston: Pearson.

[3] Hakim S, Rachmad, dan Ir. Sutarto, Msi. 2009. *Java Software Solutions Foundations of Program Design 8th Edition*. Jakarta: PT. Elex Media Komputindo.